Table of Contents

Summary

Golang 101 hacks

This is an ongoing rudimentary `Go` programming language tutorial, and it will be updated non-periodically.

Project homepage

https://github.com/NanXiao/golang-101-hacks

License

How to build Go development environment

Build `Go` development environment is always easy. Take `Linux` OS as an example (Because I work as a root user, so if you login as a non-root user, maybe you need `sudo` to execute some commands), what you should do is just download the binary package which matches your system from here, and uncompress it:

```
# wget https://storage.googleapis.com/golang/go1.6.2.linux-amd64.tar.gz
# tar -C /usr/local/ -xzf go1.6.2.linux-amd64.tar.gz
```

Now, there is an extra `go` directory under `/usr/local`. It's done! Too easy, right? Yes, but there are still some windup work to do:

(1) To run `Go` utilities (`go` , `gofmt` , etc) conveniently, you should add `/usr/local/go` into `$PATH` environment variable:

```
# cat /etc/profile
......
PATH=$PATH:/usr/local/go/bin
export PATH
......
```

(2) It is **strongly recommended** to install `Go` in `/usr/local/go` under `*nix` and `c:\Go` under `Windows` since these default directories have already been embedded in `Go` binary distributions. If you choose another directory, you **must** set `$GOROOT` environment variable:

```
# cat /etc/profile
......
GOROOT=/path/to/go
export GOROOT
```

So the `$GOROOT` is only need when you install `Go` on a custom directory, not default.

References:
Getting Started;
You don't need to set GOROOT, really.

Create Go workspace

Once the `Go` build environment is ready, the next step is to create workspace for development:

(1) Set up a new empty directory:

```
# mkdir gowork
```

(2) Use a new environment variable `$GOPATH` to point it:

```
# cat /etc/profile
......
GOPATH=/root/gowork
export GOPATH
......
```

The workspace should contain `3` subdirectories:

> src: contains the Go source code.
> pkg: contains the package objects. You could think them as libraries which are used in linkage stage to generate the final executable files.
> bin: contains the executable files.

Let's see an example:

(1) Create a `src` directory in `$GOPATH`, which is `/root/gowork` in my system:

```
# mkdir src
# tree
.
└── src

1 directory, 0 files
```

(2) Since `Go` organizes source code using " `package` " concept , and every " `package` " should occupy a distinct directory, I create a `greet` directory in `src` :

```
# mkdir src/greet
```

Then create a new `Go` source code file (`greet.go`) in `src/greet` :

```
# cat src/greet/greet.go
package greet

import "fmt"

func Greet() {
        fmt.Println("Hello 中国!")
}
```

You can consider this `greet` directory provides a `greet` package which can be used by other programs.

(3) Create another package `hello` which utilizes the `greet` package:

```
# mkdir src/hello
# cat src/hello/hello.go
package main

import "greet"

func main() {
        greet.Greet()
}
```

You can see in `hello.go` , the `main` function calls `Greet` function offered by `greet` package.

(4) Now our `$GOPATH` layout is like this:

```
# tree
.
└── src
    ├── greet
    │   └── greet.go
    └── hello
        └── hello.go

3 directories, 2 files
```

Let's compile and install `hello` package:

```
# go install hello
```

Check the `$GOPATH` layout again:

```
# tree
.
├── bin
│   └── hello
├── pkg
│   └── linux_amd64
│       └── greet.a
└── src
    ├── greet
    │   └── greet.go
    └── hello
        └── hello.go

6 directories, 4 files
```

You can see the executable command `hello` is generated in `bin` folder. Because `hello` needs `greet` package's help, a `greet.a` object is also produced in `pkg` directory, but in system related subdirectory: `linux_amd64` .

Run `hello` command:

```
# ./bin/hello
Hello 中国!
```

Working as expected!

(5) You should add `$GOPATH/bin` to `$PATH` environment variable for facility:

```
PATH=$PATH:$GOPATH/bin
export PATH
```

Then you can run `hello` directly:

```
# hello
Hello 中国!
```

Reference:
How to Write Go Code.

Package

In `Go`, the packages can be divided into `2` categories:

(1) `main` package: is used to generate the executable binary, and the `main` function is the entry point of the program. Take `hello.go` as an example:

```
package main

import "greet"

func main() {
    greet.Greet()
}
```

(2) This category can also include `2` types:

a) Library package: is used to generate the object files that can be reused by others. Take `greet.go` as an example:

```
package greet

import "fmt"

func Greet() {
    fmt.Println("Hello 中国!")
}
```

b) Some other packages for special purposes, such as testing.

Nearly every program needs `Go` standard (`$GOROOT`) or third-pary (`$GOPATH`) packages. To use them, you should use `import` statement:

```
import "fmt"
import "github.com/NanXiao/stack"
```

Or:

```
import (
    "fmt"
    "github.com/NanXiao/stack"
)
```

In the above examples, the " `fmt` " and " `github.com/NanXiao/stack` " are called `import path`, which is used to find the relevant package.

You may also see the following cases:

```
import m "lib/math" // use m as the math package name
import . "lib/math" // Omit package name when using math package
```

If the `go install` command can't find the specified package, it will complain the error messages like this:

```
... : cannot find package "xxxx" in any of:
        /usr/local/go/src/xxxx (from $GOROOT)
        /root/gowork/src/xxxx (from $GOPATH)
```

To avoid library conflicts, you'd better make your own packages' path the only one in the world: E.g., your `github` repository destination:

```
 github.com/NanXiao/...
```

Conventionally, your package name should be same with the last item in `import path`; it is a good coding habit though not a must.

Reference:
The Go Programming Language.

“go build” and “go install”

Let's tidy up the `$GOPATH` directory and only keep `Go` source code files left over:

```
# tree
.
├── bin
├── pkg
└── src
    ├── greet
    │   └── greet.go
    └── hello
        └── hello.go

5 directories, 2 files
```

The `greet.go` is `greet` package which just provides one function:

```
# cat src/greet/greet.go
package greet

import "fmt"

func Greet() {
        fmt.Println("Hello 中国!")
}
```

While `hello.go` is a `main` package which takes advantage of `greet` and can be built into an executable binary:

```
# cat src/hello/hello.go
package main

import "greet"

func main() {
        greet.Greet()
}
```

(1) Enter the `src/hello` directory, and execute `go build` command:

```
# pwd
/root/gowork/src/hello
# go build
# ls
hello  hello.go
# ./hello
Hello 中国!
```

We can see a fresh `hello` command is created in the current directory.

Check the `$GOPATH` directory:

```
# tree
.
├── bin
├── pkg
└── src
    ├── greet
    │   └── greet.go
    └── hello
        ├── hello
        └── hello.go

5 directories, 3 files
```

Compared before executing `go build`, there is only a final executable command more.

(2) Clear the `$GOPATH` directory again:

```
# tree
.
├── bin
├── pkg
└── src
    ├── greet
    │   └── greet.go
    └── hello
        └── hello.go

5 directories, 2 files
```

Running `go install` in `hello` directory:

```
# pwd
/root/gowork/src/hello
# go install
#
```

Check the `$GOPATH` directory now:

```
# tree
.
├── bin
│   └── hello
├── pkg
│   └── linux_amd64
│       └── greet.a
└── src
    ├── greet
    │   └── greet.go
    └── hello
        └── hello.go

6 directories, 4 files
```

Not only the `hello` command is generated and put into `bin` directory, but also the `greet.a` is in the `pkg/linux_amd64`. While the `src` folder keeps clean with only source code files in it and unchanged.

(3) There is `-i` flag in `go build` command which will install the packages that are dependencies of the target, but won't install the target. Let's check it:

```
# tree
.
├── bin
├── pkg
└── src
    ├── greet
    │   └── greet.go
    └── hello
        └── hello.go

5 directories, 2 files
```

Run `go build -i` under `hello` directory:

```
# pwd
#/root/gowork/src/hello
# go build -i
```

Check `$GOPATH` now:

```
# tree
.
├── bin
├── pkg
│   └── linux_amd64
│       └── greet.a
└── src
    ├── greet
    │   └── greet.go
    └── hello
        ├── hello
        └── hello.go
```

Except a `hello` command in `src/hello` directory, a `greet.a` library is also generated in `pkg/linux_amd64` too.

(4) By default, the `go build` uses the directory's name as the compiled binary's name, to modify it, you can use `-o` flag:

```
# pwd
/root/gowork/src/hello
# go build -o he
# ls
he  hello.go
```

Now, the command is `he`, not `hello`.

"go get" command

"`go get`" command is the standard way of downloading and installing packages and related dependencies, and let's check the particulars of it through an example:
(1) Create a playstack repository in github;
(2) There is a `LICENSE` file and `play` directory in `playstack` folder;
(3) The `play` directory includes one `main.go` file:

```
package main

import (
    "fmt"
    "github.com/NanXiao/stack"
)

func main() {
    s := stack.New()
    s.Push(0)
    s.Push(1)
    s.Pop()
    fmt.Println(s)
}
```

The `main` package has one dependency package: stack. Actually, the `main()` function doesn't play anything meaningful, and we just use this project as a sample. So the directory structure of `playstack` looks like this:

```
.
├── LICENSE
└── play
    └── main.go

1 directory, 2 files
```

Clean the `$GOPATH` directory, and use "`go get`" command to download `playstack`:

```
# tree
.

0 directories, 0 files
# go get github.com/NanXiao/playstack
package github.com/NanXiao/playstack: no buildable Go source files in /root/gocode/src
/github.com/NanXiao/playstack
```

" `go get` " command complains " `no buildable Go source files in ...` ", and it is because the objects which " `go get` " works are **packages**, not **repositories**.There is no `*.go` source files in `playstack` , so it is not a valid package.

Tidy up `$GOPATH` folder, and execute " `go get github.com/NanXiao/playstack/play` " instead:

```
# tree
.

0 directories, 0 files
# go get github.com/NanXiao/playstack/play
# tree
.
├── bin
│   └── play
├── pkg
│   └── linux_amd64
│       └── github.com
│           └── NanXiao
│               └── stack.a
└── src
    └── github.com
        └── NanXiao
            ├── playstack
            │   ├── LICENSE
            │   └── play
            │       └── main.go
            └── stack
                ├── LICENSE
                ├── README.md
                ├── stack.go
                └── stack_test.go

11 directories, 8 files
```

We can see not only `playstack` and its dependency (`stack`) are all downloaded, but also the command (`play`) and library (`stack`) are all installed in the right place.

The mechanism behind " `go get` " command is it will fetch the repositories of packages and dependencies (E.g., use " `git clone` ".), and you can check the detailed workflow by " `go get -x` ":

```
# tree
.

0 directories, 0 files
# go get -x github.com/NanXiao/playstack/play
cd .
git clone https://github.com/NanXiao/playstack /root/gocode/src/github.com/NanXiao/pla
ystack
cd /root/gocode/src/github.com/NanXiao/playstack
git submodule update --init --recursive
cd /root/gocode/src/github.com/NanXiao/playstack
git show-ref
cd /root/gocode/src/github.com/NanXiao/playstack
git submodule update --init --recursive
cd .
git clone https://github.com/NanXiao/stack /root/gocode/src/github.com/NanXiao/stack
cd /root/gocode/src/github.com/NanXiao/stack
git submodule update --init --recursive
cd /root/gocode/src/github.com/NanXiao/stack
git show-ref
cd /root/gocode/src/github.com/NanXiao/stack
git submodule update --init --recursive
WORK=/tmp/go-build054180753
mkdir -p $WORK/github.com/NanXiao/stack/_obj/
mkdir -p $WORK/github.com/NanXiao/
cd /root/gocode/src/github.com/NanXiao/stack
/usr/local/go/pkg/tool/linux_amd64/compile -o $WORK/github.com/NanXiao/stack.a -trimpa
th $WORK -p github.com/NanXiao/stack -complete -buildid de4d90fa03d8091e075c1be9952d69
1376f8f044 -D _/root/gocode/src/github.com/NanXiao/stack -I $WORK -pack ./stack.go
mkdir -p /root/gocode/pkg/linux_amd64/github.com/NanXiao/
mv $WORK/github.com/NanXiao/stack.a /root/gocode/pkg/linux_amd64/github.com/NanXiao/st
ack.a
mkdir -p $WORK/github.com/NanXiao/playstack/play/_obj/
mkdir -p $WORK/github.com/NanXiao/playstack/play/_obj/exe/
cd /root/gocode/src/github.com/NanXiao/playstack/play
/usr/local/go/pkg/tool/linux_amd64/compile -o $WORK/github.com/NanXiao/playstack/play.
a -trimpath $WORK -p main -complete -buildid e9a3c02979f7c6e57ce4452278c52e3e0e6a48e8
-D _/root/gocode/src/github.com/NanXiao/playstack/play -I $WORK -I /root/gocode/pkg/li
nux_amd64 -pack ./main.go
cd .
/usr/local/go/pkg/tool/linux_amd64/link -o $WORK/github.com/NanXiao/playstack/play/_ob
j/exe/a.out -L $WORK -L /root/gocode/pkg/linux_amd64 -extld=gcc -buildmode=exe -buildi
d=e9a3c02979f7c6e57ce4452278c52e3e0e6a48e8 $WORK/github.com/NanXiao/playstack/play.a
mkdir -p /root/gocode/bin/
mv $WORK/github.com/NanXiao/playstack/play/_obj/exe/a.out /root/gocode/bin/play
```

From the above output, we can see `playstack` repository is cloned first, then `stack` , At last the compilation and installation are executed.

If you only want to download the source files, and not compile and install, using " `go get -d` " command:

```
# tree
.

0 directories, 0 files
# go get -d github.com/NanXiao/playstack/play
# tree
.
└── src
    └── github.com
        └── NanXiao
            ├── playstack
            │   ├── LICENSE
            │   └── play
            │       └── main.go
            └── stack
                ├── LICENSE
                ├── README.md
                ├── stack.go
                └── stack_test.go

6 directories, 6 files
```

You can also use " `go get -u` " to update packages and their dependencies.

Reference:
Command go;
How does "go get" command know which files should be downloaded?.

Use govendor to implement vendoring

The meaning of vendoring in `Go` is squeezing a project's all dependencies into its `vendor` directory. Since `Go 1.6`, if there is a `vendor` directory in current package or its parent's directory, the dependency will be searched in `vendor` directory **first**. Govendor is such a tool to help you make use of the `vendor` feature. In the following example, I will demonstrate how to use `govendor` step by step:

(1) To be more clear, I clean `$GOPATH` folder first:

```
# tree
.

0 directories, 0 files
```

(2) I still use playstack project to do a demo, download it:

```
# go get github.com/NanXiao/playstack/play
# tree
.
├── bin
│   └── play
├── pkg
│   └── linux_amd64
│       └── github.com
│           └── NanXiao
│               └── stack.a
└── src
    └── github.com
        └── NanXiao
            ├── playstack
            │   ├── LICENSE
            │   └── play
            │       └── main.go
            └── stack
                ├── LICENSE
                ├── README.md
                ├── stack.go
                └── stack_test.go

11 directories, 8 files
```

The `playstack` depends on another 3rd-party package: stack.

(3) Install `govendor`:

```
# go get -u github.com/kardianos/govendor
```

(4) Change to `playstack` directory, and run "`govendor init`" command:

```
# cd src/github.com/NanXiao/playstack/
# govendor init
# tree
.
├── LICENSE
├── play
│   └── main.go
└── vendor
    └── vendor.json

2 directories, 3 files
```

You can see there is an additional `vendor` folder which contains `vendor.json` file:

```
# cat vendor/vendor.json
{
        "comment": "",
        "ignore": "test",
        "package": [],
        "rootPath": "github.com/NanXiao/playstack"
}
```

(5) Execute "`govendor add +external`" command:

```
# govendor add +external
# tree
.
├── LICENSE
├── play
│   └── main.go
└── vendor
    ├── github.com
    │   └── NanXiao
    │       └── stack
    │           ├── LICENSE
    │           ├── README.md
    │           └── stack.go
    └── vendor.json
```

Yeah, the `stack` project is copied to `vendor` directory now. Look at `vendor/vendor.json` file again:

```
# cat vendor/vendor.json
{
        "comment": "",
        "ignore": "test",
        "package": [
                {
                        "checksumSHA1": "3v5ClsvqF5lU/3E3c+1gf/zVeK0=",
                        "path": "github.com/NanXiao/stack",
                        "revision": "bfb214dbdb387d1c561b3b6f305ee0d8444c864b",
                        "revisionTime": "2016-04-01T05:28:44Z"
                }
        ],
        "rootPath": "github.com/NanXiao/playstack"
}
```

The `stack` package info has been updated in `vendor/vendor.json` file.

Notice: " `govendor add` " copies packages from `$GOPATH`, and you can use " `govendor fetch` " to download packages from network. You can verify it through removing `stack` package in `$GOPATH`, and execute " `govendor fetch github.com/NanXiao/stack` " command.

(6) Update `playstack` in `github`:

NanXiao Add vendoring feature.		Latest commit 8a27eb4 9 minutes ago
play	Rename main.go to play/main.go.	3 months ago
vendor	Add vendoring feature.	9 minutes ago
LICENSE	Initial commit	3 months ago

This time, clean `$GOPATH` folder and run " `go get github.com/NanXiao/playstack/play` " again:

```
# go get github.com/NanXiao/playstack/play
# tree
.
├── bin
│   └── play
├── pkg
│   └── linux_amd64
│       └── github.com
│           └── NanXiao
│               └── playstack
│                   └── vendor
│                       └── github.com
│                           └── NanXiao
│                               └── stack.a
└── src
    └── github.com
        └── NanXiao
            └── playstack
                ├── LICENSE
                ├── play
                │   └── main.go
                └── vendor
                    ├── github.com
                    │   └── NanXiao
                    │       └── stack
                    │           ├── LICENSE
                    │           ├── README.md
                    │           └── stack.go
                    └── vendor.json

18 directories, 8 files
```

Compared to previous case, it is no need to store `stack` in `$GOPATH/src/github.com/NanXiao` directory, since `playstack` has embedded it in its `vendor` folder.

This is just a simple intro of `govendor`, for more commands' usages, you should visit its project home page.

Reference:
What does the term “vendoring” or “to vendor” mean for Ruby on Rails?;
Understanding and using the vendor folder;
Go Vendoring Beginner Tutorial.

init function

There is a `init()` function, as the name suggests, it will do some initialization work, such as initializing variables which may not be expressed easily, or calibrating program state. A file can contain one or more `init()` functions, as shown here:

```
package main

import "fmt"

var global int = 0

func init() {
    global++
    fmt.Println("In first Init(), global is: ", global)
}

func init() {
    global++
    fmt.Println("In Second Init(), global is: ", global)
}

func main() {
    fmt.Println("In main(), global is: ", global)
}
```

The execution result is like this:

```
In first Init(), global is:  1
In Second Init(), global is:  2
In main(), global is:  2
```

Since one package can contain multiple files, there may be many `init()` functions. You **should not** presume which file's `init()` functions are executed first. The only thing which is guaranteed is that the variables declared in package will be evaluated before all `init()` functions are executed in this package.

See another example. The `$GOROOT/src` directory is like this:

```
# tree
.
├── foo
│   └── foo.go
└── play
    └── main.go
```

There are `2` simple packages: `foo` and `play`. The `foo/foo.go` is here:

```
package foo

import "fmt"

var Global int

func init() {
        Global++
        fmt.Println("foo init() is called, Global is: ", Global)
}
```

While the `play/main.go` is:

```
package main

import "foo"


func main() {
}
```

Build `play` command:

```
# go install play
# play
src/play/main.go:3: imported and not used: "foo"
```

The cause of this error is that `main.go` doesn't use any functions or variables exported by `foo` package. So if you just want an imported package's `init()` function is executed, and don't want to use package's other stuff, you should modify " `import "foo"` " to " `import _ "foo"` ":

```
 package main

import _ "foo"


func main() {
}
```

Now the build process will success, and the output of `play` command is like this:

```
# play
foo init() is called, Global is:  1
```

References:
Effective Go;
When is the init() function in go (golang) run?.

Short variable declaration

Short variable declaration is a very convenient manner of "declaring variable" in `Go`:

```
i := 10
```

It is shorthand of following (Please notice there is no type):

```
var i = 10
```

The `Go` compiler will infer the type according to the value of variable. It is a very handy feature, but on the other side of coin, it also brings some pitfalls which you should pay attention to:

(1) This format can only be used in functions:

```
package main

i := 10

func main() {
    fmt.Println(i)
}
```

The compiler will complain the following words:

```
syntax error: non-declaration statement outside function body
```

(2) You must declare **at least 1 new variable**:

```
package main

import "fmt"

func main() {
    var i = 1

    i, err := 2, true

    fmt.Println(i, err)
}
```

In `i, err := 2, false` statement, only `err` is a new declared variable, `var` is actually declared in `var i = 1`.

(3) The short variable declaration can shadow the global variable declaration, and it may not be what you want, and gives you a big surprise:

```
package main

import "fmt"

var i = 1

func main() {

    i, err := 2, true

    fmt.Println(i, err)
}
```

`i, err := 2, true` actually declares a **new local i** which makes the **global i** inaccessible in `main` function. To use the global variable but not introducing a new local one, one solution maybe like this:

```
package main

import "fmt"

var i int

func main() {

    var err bool

    i, err = 2, true

    fmt.Println(i, err)
}
```

Reference：
Short variable declarations.

“nil slice” vs “nil map”

Slice and map are all reference types in `Go`, and their default values are `nil`:

```
package main

import "fmt"

func main() {
    var (
        s []int
        m map[int]bool
    )
    if s == nil {
        fmt.Println("The value of s is nil")
    }
    if m == nil {
        fmt.Println("The value of m is nil")
    }
}
```

The result is like this：

```
The value of s is nil
The value of m is nil
```

When a slice's value is `nil`, you could also do operations on it, such as `append`:

```
package main

import "fmt"

func main() {
    var s []int
    fmt.Println("Is s a nil? ", s == nil)
    s = append(s, 1)
    fmt.Println("Is s a nil? ", s == nil)
    fmt.Println(s)
}
```

The result is like this：

```
Is s a nil?  true
Is s a nil?  false
[1]
```

A caveat you should notice is the length of both `nil` and empty slice is `0`:

```
package main

import "fmt"

func main() {
    var s1 []int
    s2 := []int{}
    fmt.Println("Is s1 a nil? ", s1 == nil)
    fmt.Println("Length of s1 is: ", len(s1))
    fmt.Println("Is s2 a nil? ", s2 == nil)
    fmt.Println("Length of s2 is: ", len(s2))
}
```

The result is like this：

```
Is s1 a nil?  true
Length of s1 is:  0
Is s2 a nil?  false
Length of s2 is:  0
```

So you should compare the slice's value with `nil` to check whether it is a `nil`.

Accessing a `nil` map is OK, but storing a `nil` map cause program panic:

```
package main

import "fmt"

func main() {
    var m map[int]bool
    fmt.Println("Is m a nil? ", m == nil)
    fmt.Println("m[1] is ", m[1])
    m[1] = true
}
```

The result is like this:

```
Is m a nil?  true
m[1] is  false
panic: assignment to entry in nil map

goroutine 1 [running]:
panic(0x4cc0e0, 0xc082034210)
    C:/Go/src/runtime/panic.go:481 +0x3f4
main.main()
    C:/Work/gocode/src/Hello.go:9 +0x2ee
exit status 2

Process finished with exit code 1
```

So the best practice is to initialize a `map` before using it, like this:

```
m := make(map[int]bool)
```

BTW, you should use the following pattern to check whether there is an element in map or not:

```
if v, ok := m[1]; !ok {
    .....
}
```

Reference:
The Go Programming Language.

Prepend

`Go` has a built-in append function which add elements in the slice:

```
func append(slice []Type, elems ...Type) []Type
```

But how if we want to the "prepend" effect? Maybe we should use `copy` function. E.g.:

```
package main

import "fmt"

func main()  {
    var s []int = []int{1, 2}
    fmt.Println(s)

    s1 := make([]int, len(s) + 1)
    s1[0] = 0
    copy(s1[1:], s)
    s = s1
    fmt.Println(s)

}
```

The result is like this:

```
[1 2]
[0 1 2]
```

But the above code looks ugly and cumbersome, so an elegant implementation maybe here:

```
s = append([]int{0}, s...)
```

BTW, I also have tried to write a "general-purpose" prepend:

```
func Prepend(v interface{}, slice []interface{}) []interface{}{
    return append([]interface{}{v}, slice...)
}
```

But since `[]T` can't convert to an `[]interface{}` directly (please refer https://golang.org/doc/faq#convert_slice_of_interface, it is just a toy, not useful.

Reference:
Go – append/prepend item into slice.

String

In `Go`, string is an immutable array of bytes. So if created, we can't change its value. E.g.:

```
package main

func main()  {
    s := "Hello"
    s[0] = 'h'
}
```

The compiler will complain:

```
cannot assign to s[0]
```

To modify the content of a string, you could convert it to a `byte` array. But in fact, you **do not** operate on the original string, just a copy:

```
package main

import "fmt"

func main()  {
    s := "Hello"
    b := []byte(s)
    b[0] = 'h'
    fmt.Printf("%s\n", b)
}
```

The result is like this:

```
hello
```

Since `Go` uses `UTF-8` encoding, you must remember the `len` function will return the string's byte number, not character number:

```
package main

import "fmt"

func main()  {
    s := "日志log"
    fmt.Println(len(s))
}
```

The result is:

```
9
```

Because each Chinese character occupied `3` bytes, `s` in the above example contains `5` characters and `9` bytes.

If you want to access every character, `for ... range` loop can give a help:

```
package main
import "fmt"

func main() {
    s := "日志log"
    for index, runeValue := range s {
        fmt.Printf("%#U starts at byte position %d\n", runeValue, index)
    }
}
```

The result is:

```
U+65E5 '日' starts at byte position 0
U+5FD7 '志' starts at byte position 3
U+006C 'l' starts at byte position 6
U+006F 'o' starts at byte position 7
U+0067 'g' starts at byte position 8
```

Reference:
Strings, bytes, runes and characters in Go;
The Go Programming Language.

The internals of slice

There are `3` components of slice:
a) `Pointer`: Points to the start position of slice in the underlying array;
b) `length` (type is `int`): the number of the valid elements of the slice;
b) `capacity` (type is `int`): the total number of slots of the slice.

Check the following code:

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    var s1 []int
    fmt.Println(unsafe.Sizeof(s1))
}
```

The result is `24` on my `64-bit` system (The `pointer` and `int` both occupy `8` bytes).

In the next example, I will use `gdb` to poke the internals of slice. The code is like this:

```
package main

import "fmt"

func main() {
        s1 := make([]int, 3, 5)
        copy(s1, []int{1, 2, 3})
        fmt.Println(len(s1), cap(s1), &s1[0])

        s1 = append(s1, 4)
        fmt.Println(len(s1), cap(s1), &s1[0])

        s2 := s1[1:]
        fmt.Println(len(s2), cap(s2), &s2[0])
}
```

Use `gdb` to step into the code:

```
5       func main() {
(gdb) n
6               s1 := make([]int, 3, 5)
(gdb)
7               copy(s1, []int{1, 2, 3})
(gdb)
8               fmt.Println(len(s1), cap(s1), &s1[0])
(gdb)
3 5 0xc820010240
```

Before executing " `s1 = append(s1, 4)` ", `fmt.Println` outputs the length(`3`), capacity(`5`) and the starting element address(`0xc820010240`) of the slice, let's check the memory layout of `s1` :

```
10              s1 = append(s1, 4)
(gdb) p &s1
$1 = (struct []int *) 0xc82003fe40
(gdb) x/24xb 0xc82003fe40
0xc82003fe40:   0x40    0x02    0x01    0x20    0xc8    0x00    0x00    0x00
0xc82003fe48:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc82003fe50:   0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

Through examining the memory content of `s1` (the start memory address is `0xc82003fe40`), we can see its content matches the output of `fmt.Println` .

Continue executing, and check the result before " `s2 := s1[1:]` ":

```
(gdb) n
11              fmt.Println(len(s1), cap(s1), &s1[0])
(gdb)
4 5 0xc820010240
13              s2 := s1[1:]
(gdb) x/24xb 0xc82003fe40
0xc82003fe40:   0x40    0x02    0x01    0x20    0xc8    0x00    0x00    0x00
0xc82003fe48:   0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc82003fe50:   0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

We can see after appending a new element(`s1 = append(s1, 4)`), the length of `s1` is changed to `4` , but the capacity remains the original value.

Let's check the internals of `s2` :

```
(gdb) n
14              fmt.Println(len(s2), cap(s2), &s2[0])
(gdb)
3 4 0xc820010248
15      }
(gdb) p &s2
$3 = (struct []int *) 0xc82003fe28
(gdb) x/24hb 0xc82003fe28
0xc82003fe28:   0x48    0x02    0x01    0x20    0xc8    0x00    0x00    0x00
0xc82003fe30:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc82003fe38:   0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

The element start address of `s2` is `0xc820010248`, actually the second element of `s1` (`0xc82003fe40`), and the length(`3`) and capacity(`4`) are both one less than the counterparts of `s1` (`4` and `5` respectively).

Pass slice as a function argument

In `Go`, the function parameters are passed by value. With respect to use slice as a function argument, that means the function will get the copies of the slice: a pointer which points to the starting address of the underlying array, accompanied by the length and capacity of the slice. Oh boy! Since you know the address of the memory which is used to store the data, you can tweak the slice now. Let's see the following example:

```
package main

import (
    "fmt"
)

func modifyValue(s []int)  {
    s[1] = 3
    fmt.Printf("In modifyValue: s is %v\n", s)
}
func main() {
    s := []int{1, 2}
    fmt.Printf("In main, before modifyValue: s is %v\n", s)
    modifyValue(s)
    fmt.Printf("In main, after modifyValue: s is %v\n", s)
}
```

The result is here:

```
In main, before modifyValue: s is [1 2]
In modifyValue: s is [1 3]
In main, after modifyValue: s is [1 3]
```

You can see, after running `modifyValue` function, the content of slice `s` is changed. Although the `modifyValue` function just gets a copy of the memory address of slice's underlying array, it is enough!

See another example:

```
package main

import (
    "fmt"
)

func addValue(s []int) {
    s = append(s, 3)
    fmt.Printf("In addValue: s is %v\n", s)
}

func main() {
    s := []int{1, 2}
    fmt.Printf("In main, before addValue: s is %v\n", s)
    addValue(s)
    fmt.Printf("In main, after addValue: s is %v\n", s)
}
```

The result is like this:

```
In main, before addValue: s is [1 2]
In addValue: s is [1 2 3]
In main, after addValue: s is [1 2]
```

This time, the `addValue` function doesn't take effect on the `s` slice in `main` function. That's because it just manipulate the copy of the `s`, not the "real" `s`.

So if you really want the function to change the content of a slice, you can pass the address of the slice:

```
package main

import (
    "fmt"
)

func addValue(s *[]int) {
    *s = append(*s, 3)
    fmt.Printf("In addValue: s is %v\n", s)
}

func main() {
    s := []int{1, 2}
    fmt.Printf("In main, before addValue: s is %v\n", s)
    addValue(&s)
    fmt.Printf("In main, after addValue: s is %v\n", s)
}
```

The result is like this:

```
In main, before addValue: s is [1 2]
In addValue: s is &[1 2 3]
In main, after addValue: s is [1 2 3]
```

Two-dimensional slice

`Go` supports multiple-dimensional slice, but I only want to introduce two-dimensional slice here. One reason is the two-dimensional slice is usually used in daily life, while multiple-dimensional seems not common. If you often use multiple-dimensional slice, personally I think the code is a little clumsy and not easy to maintain, so maybe you can try to check whether there is a better method; the other reason is the principle behind multiple-dimensional slice is the same with two-dimensional slice, you can also understand it if you know two-dimensional slice well.

Let's the following example:

```
package main

import "fmt"

func main() {
    s := make([][]int, 2)
    fmt.Println(len(s), cap(s), &s[0])

    s[0] = []int{1, 2, 3}
    fmt.Println(len(s[0]), cap(s[0]), &s[0][0])

    s[1] = make([]int, 3, 5)
    fmt.Println(len(s[1]), cap(s[1]), &s[1][0])
}
```

I still use gdb to inspect the execution flow:

```
5       func main() {
(gdb) n
6               s := make([][]int, 2)
(gdb)
7               fmt.Println(len(s), cap(s), &s[0])
(gdb)
2 2 &[]
9               s[0] = []int{1, 2, 3}
(gdb) p &s
$1 = (struct [][]int *) 0xc82003fe70
(gdb) x/24xb 0xc82003fe70
0xc82003fe70:   0x40    0x02    0x01    0x20    0xc8    0x00    0x00    0x00
0xc82003fe78:   0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc82003fe80:   0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

`s` is a slice (the start memory address is `0xc82003fe70`), but its elements are also slices. Let's check the elements:

```
(gdb) x/48xb 0xc820010240
0xc820010240:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010248:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010250:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010258:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010260:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010268:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

All the memory content are `0`, nothing exciting! Continue to step by step:

```
(gdb) n
10              fmt.Println(len(s[0]), cap(s[0]), &s[0][0])
(gdb)
3 3 0xc82000e220
12              s[1] = make([]int, 3, 5)
```

Now since `s` contains a valid slice element, check its underlying array:

```
(gdb) x/48xb 0xc820010240
0xc820010240:   0x20    0xe2    0x00    0x20    0xc8    0x00    0x00    0x00
0xc820010248:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010250:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010258:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010260:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010268:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Yeah, the memory has been updated by the pointer, length and capacity of `s[0]`, the same with previous output from `fmt.Println`. Check the underlying array of `s[0]`:

```
(gdb) x/24xb 0xc82000e220
0xc82000e220:   0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc82000e228:   0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc82000e230:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

We can see `3` elements: `1`, `2`, `3`.

Following the same method to check the `s[1]`:

```
(gdb) n
13              fmt.Println(len(s[1]), cap(s[1]), &s[1][0])
(gdb)
3 5 0xc820010270
14      }
(gdb) x/48xb 0xc820010240
0xc820010240:   0x20    0xe2    0x00    0x20    0xc8    0x00    0x00    0x00
0xc820010248:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010250:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010258:   0x70    0x02    0x01    0x20    0xc8    0x00    0x00    0x00
0xc820010260:   0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010268:   0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/40xb 0xc820010270
0xc820010270:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010278:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010280:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010288:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xc820010290:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Now, we can see `s` contains all the info of its slice elements, and the elements of `s[1]` are initialized to `0`.

Reallocating underlying array of slice

When appending data into slice, if the underlying array of the slice doesn't have enough space, a new array will be allocated. Then the elements in old array will be copied into this new memory, accompanied with adding new data behind. So when using `Go` built-in `append` function, you must always keep the idea that "the array may have been changed" in mind, and be very careful about it, otherwise, it may bite you!

Let me explain it through a contrived example:

```
package main

import (
    "fmt"
)

func addTail(s []int)  {
    var ns [][]int
    for _, v := range []int{1, 2} {
        ns = append(ns, append(s, v))
    }
    fmt.Println(ns)
}

func main() {
    s1 := []int{0, 0}
    s2 := append(s1, 0)

    for _, v := range [][]int{s1, s2} {
        addTail(v)
    }
}
```

The `s1` is `[0, 0]`, and the `s2` is `[0, 0, 0]`; in `addTail` function, I want to add `1` and `2` behind the slice. So the wanted output is like this:

```
[[0 0 1] [0 0 2]]
[[0 0 0 1] [0 0 0 2]]
```

But the actual result is:

```
[[0 0 1] [0 0 2]]
[[0 0 0 2] [0 0 0 2]]
```

The operations on `s1` are successful, while `s2` not.

Let's use `delve` to debug this issue and check the internal mechanism of slice: Add breakpoint on `addTail` function, and it is first hit when processing `s1`:

```
(dlv) n
> main.addTail() ./slice.go:8 (PC: 0x401022)
     3: import (
     4:         "fmt"
     5: )
     6:
     7: func addTail(s []int)  {
=>   8:         var ns [][]int
     9:         for _, v := range []int{1, 2} {
    10:                 ns = append(ns, append(s, v))
    11:         }
    12:         fmt.Println(ns)
    13: }
(dlv) p s
[]int len: 2, cap: 2, [0,0]
(dlv) p &s[0]
(*int)(0xc82000a2a0)
```

The length and capacity of `s1` are both `2`, and the underlying array address is `0xc82000a2a0`, so what happened when executing the following statement:

```
ns = append(ns, append(s, v))
```

Since the length and capacity of `s1` are both `2`, there is no room for new buddy. To append a new value, a new array must be allocated, and it contains both `[0, 0]` from `s1` and the new value(`1` or `2`). You can consider `append(s, v)` generated an anonymous new slice, and it is appended in `ns`. We can check it after running " `ns = append(ns, append(s, v))` ":

```
(dlv) n
> main.addTail() ./slice.go:9 (PC: 0x401217)
     4:         "fmt"
     5: )
     6:
     7: func addTail(s []int)  {
     8:         var ns [][]int
=>   9:         for _, v := range []int{1, 2} {
    10:                 ns = append(ns, append(s, v))
    11:         }
    12:         fmt.Println(ns)
    13: }
    14:
(dlv) p ns
[][]int len: 1, cap: 1, [
        [0,0,1],
]
(dlv) p ns[0]
[]int len: 3, cap: 4, [0,0,1]
(dlv) p &ns[0][0]
(*int)(0xc82000e240)
(dlv) p s
[]int len: 2, cap: 2, [0,0]
(dlv) p &s[0]
(*int)(0xc82000a2a0)
```

We can see the length of anonymous slice is `3`, capacity is `4`, and the underlying array address is `0xc82000e240`, different from `s1`'s (`0xc82000a2a0`). Continue executing until exit loop:

```
(dlv) n
> main.addTail() ./slice.go:12 (PC: 0x40124c)
     7: func addTail(s []int)  {
     8:         var ns [][]int
     9:         for _, v := range []int{1, 2} {
    10:                 ns = append(ns, append(s, v))
    11:         }
=>  12:         fmt.Println(ns)
    13: }
    14:
    15: func main() {
    16:         s1 := []int{0, 0}
    17:         s2 := append(s1, 0)
(dlv) p ns
[][]int len: 2, cap: 2, [
        [0,0,1],
        [0,0,2],
]
(dlv) p &ns[0][0]
(*int)(0xc82000e240)
(dlv) p &ns[1][0]
(*int)(0xc82000e280)
(dlv) p &s[0]
(*int)(0xc82000a2a0)
```

We can see `s1` , `ns[0]` and `ns[1]` have `3` independent array.

Now, let's follow the same steps to check what happened on `s2` :

```
(dlv) n
> main.addTail() ./slice.go:8 (PC: 0x401022)
     3: import (
     4:         "fmt"
     5: )
     6:
     7: func addTail(s []int)  {
=>   8:         var ns [][]int
     9:         for _, v := range []int{1, 2} {
    10:                 ns = append(ns, append(s, v))
    11:         }
    12:         fmt.Println(ns)
    13: }
(dlv) p s
[]int len: 3, cap: 4, [0,0,0]
(dlv) p &s[0]
(*int)(0xc82000e220)
```

The length of `s2` is `3`, and capacity is `4`, so there is one slot for adding new element. Check the `s2` and `ns`' values after executing "`ns = append(ns, append(s, v))`" the first time:

```
(dlv)
> main.addTail() ./slice.go:9 (PC: 0x401217)
     4:         "fmt"
     5: )
     6:
     7: func addTail(s []int)  {
     8:         var ns [][]int
=>   9:         for _, v := range []int{1, 2} {
    10:                 ns = append(ns, append(s, v))
    11:         }
    12:         fmt.Println(ns)
    13: }
    14:
(dlv) p ns
[][]int len: 1, cap: 1, [
        [0,0,0,1],
]
(dlv) p &ns[0][0]
(*int)(0xc82000e220)
(dlv) p s
[]int len: 3, cap: 4, [0,0,0]
(dlv) p &s[0]
(*int)(0xc82000e220)
```

We can see the new anonymous slice's array address is also `0xc82000e220`, that's because the `s2` has enough space to hold new value, no new array is allocated. Check the `s2` and `ns` again after adding `2`:

```
(dlv)
> main.addTail() ./slice.go:12 (PC: 0x40124c)
     7: func addTail(s []int)  {
     8:         var ns [][]int
     9:         for _, v := range []int{1, 2} {
    10:                 ns = append(ns, append(s, v))
    11:         }
=>  12:         fmt.Println(ns)
    13: }
    14:
    15: func main() {
    16:         s1 := []int{0, 0}
    17:         s2 := append(s1, 0)
(dlv) p ns
[][]int len: 2, cap: 2, [
        [0,0,0,2],
        [0,0,0,2],
]
(dlv) p &ns[0][0]
(*int)(0xc82000e220)
(dlv) p &ns[1][0]
(*int)(0xc82000e220)
(dlv) p s
[]int len: 3, cap: 4, [0,0,0]
(dlv) p &s[0]
(*int)(0xc82000e220)
```

All `3` slices point to the same array, so the later value(`2`) will override previous item(`1`).

So in a conclusion, `append` is very tricky since it can modify the underlying array without noticing you. You must know the memory layout behind every slice clearly, else the slice can give you a big, unwanted surprise!

copy

The definition of built-in `copy` function is here:

> func copy(dst, src []Type) int
>
> The copy built-in function copies elements from a source slice into a destination slice. (As a special case, it also will copy bytes from a string to a slice of bytes.) The source and destination may overlap. Copy returns the number of elements copied, which will be the minimum of len(src) and len(dst).

Let's see a basic example in which source and destination slices aren't overlapped:

```
package main

import (
    "fmt"
)

func main() {
    d := make([]int, 3, 5)
    s := []int{2, 2}
    fmt.Println("Before copying (destination slice): ", d)
    fmt.Println("Copy length is: ", copy(d, s))
    fmt.Println("After copying (destination slice): ", d)

    d = make([]int, 3, 5)
    s = []int{2, 2, 2}
    fmt.Println("Before copying (destination slice): ", d)
    fmt.Println("Copy length is: ", copy(d, s))
    fmt.Println("After copying (destination slice): ", d)

    d = make([]int, 3, 5)
    s = []int{2, 2, 2, 2}
    fmt.Println("Before copying (destination slice): ", d)
    fmt.Println("Copy length is: ", copy(d, s))
    fmt.Println("After copying (destination slice): ", d)

}
```

In the above example, the destination slice's length is `3`, and the source slice's length can be `2`, `3`, `4`. Check the result:

```
Before copying (destination slice):  [0 0 0]
Copy length is:  2
After copying (destination slice):  [2 2 0]
Before copying (destination slice):  [0 0 0]
Copy length is:  3
After copying (destination slice):  [2 2 2]
Before copying (destination slice):  [0 0 0]
Copy length is:  3
After copying (destination slice):  [2 2 2]
```

We can make sure the number of copied elements is indeed the minimum length of source and destination slices.

Let's check the overlapped case:

```
package main

import (
    "fmt"
)

func main() {
    d := []int{1, 2, 3}
    s := d[1:]

    fmt.Println("Before copying: ", "source is: ", s, "destination is: ", d)
    fmt.Println(copy(d, s))
    fmt.Println("After copying: ", "source is: ", s, "destination is: ", d)

    s = []int{1, 2, 3}
    d = s[1:]

    fmt.Println("Before copying: ", "source is: ", s, "destination is: ", d)
    fmt.Println(copy(d, s))
    fmt.Println("After copying: ", "source is: ", s, "destination is: ", d)
}
```

The result is like this:

```
Before copying:  source is:  [2 3] destination is:  [1 2 3]
2
After copying:  source is:  [3 3] destination is:  [2 3 3]
Before copying:  source is:  [1 2 3] destination is:  [2 3]
2
After copying:  source is:  [1 1 2] destination is:  [1 2]
```

Through the output, we can see no matter the source slice is ahead of destination or not, the result is always as expected. You can think the implementation is like this: the data from source slice are copied to a temporary place first, then the elements are copied from temporary to destination slice.

`copy` requires the source and destination slices are the same type, and an exception is the source is string while the destination is `[]byte` :

```
package main

import (
	"fmt"
)

func main() {
	d := make([]byte, 20, 30)
	fmt.Println(copy(d, "Hello, 中国"))
	fmt.Println(string(d))
}
```

The output is:

```
13
Hello, 中国
```

Reference:
copy() behavior when overlapping.

Array

In `Go`, the length is also a part of array type. So the following code declares an array:

```
var array [3]int
```

while "`var slice []int`" defines a slice. Because of this characteristic, arrays with the same array element type but different length can't assign values each other. I.E.:

```
package main

import "fmt"

func main() {
    var a1 [2]int
    var a2 [3]int
    a2 = a1
    fmt.Println(a2)
}
```

The compiler will complain:

```
cannot use a1 (type [2]int) as type [3]int in assignment
```

Changing "`var a1 [2]int`" to "`var a1 [3]int`" will make it work.

Another caveat you should pay attention to is the following code declares an array, not a slice:

```
array := [...]int {1, 2, 3}
```

You can verify it by the following code:

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    array := [...]int {1, 2, 3}
    slice := []int{1, 2, 3}
    fmt.Println(reflect.TypeOf(array), reflect.TypeOf(slice))
}
```

The output is:

```
[3]int []int
```

Additionally, since in `Go`, the function argument is passed by "value", so if you use an array as a function argument, the function just does the operations on the copy of the original copy. Check the following code:

```
package main

import (
    "fmt"
)

func changeArray(array [3]int) {
    for i, _ := range array {
        array[i] = 1
    }
    fmt.Printf("In changeArray function, array address is %p, value is %v\n", &array,
array)
}

func main() {
    var array [3]int

    fmt.Printf("Original array address is %p, value is %v\n", &array, array)
    changeArray(array)
    fmt.Printf("Changed array address is %p, value is %v\n", &array, array)
}
```

The output is:

```
Original array address is 0xc082008680, value is [0 0 0]
In changeArray function, array address is 0xc082008700, value is [1 1 1]
Changed array address is 0xc082008680, value is [0 0 0]
```

From the log, you can see the array's address in `changeArray` function is not the same with array's address in `main` function, so the content of original array will definitely not be modified. Furthermore, if the array is very large, copying them when passing argument to function may generate more overhead than you want, you should know about it.

Conversion between array and slice

In `Go`, array is a fixed length of continuous memory with specified type, while slice is just a reference which points to an underlying array. Since they are different types, they can't assign value each other directly. See the following example:

```
package main

import "fmt"

func main() {
    s := []int{1, 2, 3}
    var a [3]int

    fmt.Println(copy(a, s))
}
```

Because `copy` only accepts slice argument, we can use the `[:]` to create a slice from array. Check next code:

```
package main

import "fmt"

func main() {
    s := []int{1, 2, 3}
    var a [3]int

    fmt.Println(copy(a[:2], s))
    fmt.Println(a)
}
```

The running output is:

```
2
[1 2 0]
```

The above example is copying value from slice to array, and the opposite operation is similar:

```
package main

import "fmt"

func main() {
    a := [...]int{1, 2, 3}
    s := make([]int, 3)

    fmt.Println(copy(s, a[:2]))
    fmt.Println(s)
}
```

The execution result is:

```
2
[1 2 0]
```

References:
In golang how do you convert a slice into an array;
Arrays, slices (and strings): The mechanics of 'append'.

Accessing map

Map is a reference type which points to a hash table, and you can use it to construct a "key-value" database which is very handy in practice programming. E.g., the following code will calculate the count of every element in a slice:

```
package main

import (
    "fmt"
)

func main() {
    s := []int{1, 1, 2, 2, 3, 3, 3}
    m := make(map[int]int)

    for _, v := range s {
        m[v]++
    }

    for key, value := range m {
        fmt.Printf("%d occurs %d times\n", key, value)
    }
}
```

The output is like this:

```
3 occurs 3 times
1 occurs 2 times
2 occurs 2 times
```

Moreover, according to Go spec: "A map is an **unordered** group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type.". So if you run the above program another time, the output may be different:

```
2 occurs 2 times
3 occurs 3 times
1 occurs 2 times
```

You must not presume the element order of a map.

The key type of the map must can be compared with "`==`" operator: the built-in types, such as int, string, etc, satisfy this requirement; while slice not. For struct type, if its members all can be compared by "`==`" operator, then this struct can also be used as key.

When you access a non-exist key of the map, the map will return the `nil` value of the element. I.e.:

```
package main

import (
    "fmt"
)

func main() {
    m := make(map[int]bool)

    m[0] = false
    m[1] = true

    fmt.Println(m[0], m[1], m[2])
}
```

The output is:

```
false true false
```

the value of `m[0]` and `m[2]` are both `false`, so you can't discriminate whether the key is really in map or not. The solution is to use “comma ok” method:

```
value, ok := map[key]
```

if the key does exit, `ok` will be `true`; else `ok` will be `false`.

Sometimes, you may not care the values of the map, and use map just as a set. In this case, you can declare the value type as an empty struct: `struct{}`. An example is like this:

```
package main

import (
    "fmt"
)

func check(m map[int]struct{}, k int) {
    if _, ok := m[k]; ok {
        fmt.Printf("%d is a valid key\n", k)
    }
}
func main() {
    m := make(map[int]struct{})
    m[0] = struct{}{}
    m[1] = struct{}{}

    for  i := 0; i <=2; i++ {
        check(m, i)
    }
}
```

The output is:

```
0 is a valid key
1 is a valid key
```

Using built-in `delete` function, you can remove an entry in the map, even the key doesn't exist:

```
delete(map, key)
```

References:
Effective Go;
The Go Programming Language Specification;
The Go Programming Language.

switch

Compared to other programming languages (such as `C`), `Go`'s `switch-case` statement doesn't need explicit "`break`", and not have `fall-though` characteristic. Take the following code as an example:

```
package main

import (
    "fmt"
)

func checkSwitch(val int) {
    switch val {
    case 0:
    case 1:
        fmt.Println("The value is: ", val)
    }
}
func main() {
    checkSwitch(0)
    checkSwitch(1)
}
```

The output is:

```
The value is:  1
```

Your real intention is the "`fmt.Println("The value is: ", val)`" will be executed when `val` is `0` or `1`, but in fact, the statement only takes effect when `val` is `1`. To fulfill your request, there are `2` methods:

(1) Use `fallthrough`:

```
func checkSwitch(val int) {
    switch val {
    case 0:
        fallthrough
    case 1:
        fmt.Println("The value is: ", val)
    }
}
```

(2) Put `0` and `1` in the same `case`:

```
func checkSwitch(val int) {
    switch val {
    case 0, 1:
        fmt.Println("The value is: ", val)
    }
}
```

`switch` can also be used as a better `if-else`, and you may find it may be more clearer and simpler than multiple `if-else` statements.E.g.:

```
package main

import (
    "fmt"
)

func checkSwitch(val int) {
    switch {
    case val < 0:
        fmt.Println("The value is less than zero.")
    case val == 0:
        fmt.Println("The value is qual to zero.")
    case val > 0:
        fmt.Println("The value is more than zero.")
    }
}
func main() {
    checkSwitch(-1)
    checkSwitch(0)
    checkSwitch(1)
}
```

The output is:

```
The value is less than zero.
The value is qual to zero.
The value is more than zero.
```

Interface

Interface is a reference type which contains some method definitions. Any type which implements all the methods defined by a reference type will satisfy this interface type automatically. Through interface, you can approach object-oriented programming. Check the following example:

```
package main

import "fmt"

type Foo interface {
    foo()
}

type A struct {
}

func (a A) foo() {
    fmt.Println("A foo")
}

func (a A) bar() {
    fmt.Println("A bar")
}

func callFoo(f Foo) {
    f.foo()
}

func main() {
    var a A
    callFoo(a)
}
```

The running result is:

```
A foo
```

Let's analyze the code detailedly:

(1)

```
type Foo interface {
    foo()
}
```

The above code defines a interface `Foo` which has only one method: `foo()`.

(2)

```
type A struct {
}

func (a A) foo() {
    fmt.Println("A foo")
}

func (a A) bar() {
    fmt.Println("A bar")
}
```

struct `A` has `2` methods: `foo()` and `bar()`. Since it already implements `foo()` method, it satisfies `Foo` interface.

(3)

```
func callFoo(f Foo) {
    f.foo()
}

func main() {
    var a A
    callFoo(a)
}
```

`callFoo` requires a variable whose type is `Foo` interface, and passing `A` is OK. The `callFoo` will use `A`'s `foo()` method, and "`A foo`" is printed.

Let's change the `main()` function:

```
func main() {
    var a A
    callFoo(&a)
}
```

This time, the argument of `callFoo()` is `&a`, whose type is `*A`. Compile and run the program, you may find it also outputs: "`A foo`". So `*A` type has all the methods which `A` has. But the reverse is not true:

```
package main

import "fmt"

type Foo interface {
    foo()
}

type A struct {
}

func (a *A) foo() {
    fmt.Println("A foo")
}

func (a *A) bar() {
    fmt.Println("A bar")
}

func callFoo(f Foo) {
    f.foo()
}

func main() {
    var a A
    callFoo(a)
}
```

Compile the program:

```
example.go:26: cannot use a (type A) as type Foo in argument to callFoo:
A does not implement Foo (foo method has pointer receiver)
```

You can see also `*A` type has implemented `foo()` and `bar()` methods, it doesn't mean `A` type has both methods by default.

BTW, every type satisfies the empty interface: `interface{}` .

The interface type is actually a tuple which contains `2` elements: `<type, value>` , `type` identifies the type of the variable which stores in the interface while `value` points to the actual value. The default value of an interface type is `nil` , which means both `type` and `value` are `nil` : `<nil, nil>` . When you check whether an interface is empty or not:

```
var err error
if err != nil {
    ...
}
```

You must remember only if both `type` and `value` are `nil` means the interface value is `nil`.

Reference:
The Go Programming Language.

Type assertion and type switch

Sometimes, you may want to know the exact type of an interface variable. In this scenario, you can use `type assertion`:

```
x.(T)
```

`x` is the variable whose type must be **interface**, and `T` is the type which you want to check. For example:

```
package main

import "fmt"

func printValue(v interface{}) {
    fmt.Printf("The value of v is: %v", v.(int))
}

func main() {
    v := 10
    printValue(v)
}
```

The running result is:

```
The value of v is: 10
```

In the above example, using `v.(int)` to assert the `v` is `int` variable.

if the `type assertion` operation fails, a running panic will occur: change

```
fmt.Printf("The value of v is: %v", v.(int))
```

into:

```
fmt.Printf("The value of v is: %v", v.(string))
```

Then executing the program will get following error:

```
panic: interface conversion: interface is int, not string

goroutine 1 [running]:
panic(0x4f0840, 0xc0820042c0)
......
```

To avoid this, `type assertion` actually returns an additional `boolean` variable to tell whether this operations holds or not. So modify the program as follows:

```
package main

import "fmt"

func printValue(v interface{}) {
    if v, ok := v.(string); ok {
        fmt.Printf("The value of v is: %v", v)
    } else {
        fmt.Println("Oops, it is not a string!")
    }

}

func main() {
    v := 10
    printValue(v)
}
```

This time, the output is:

```
Oops, it is not a string!
```

Furthermore, you can also use `type switch` which makes use of `type assertion` to determine the type of variable, and do the operations accordingly. Check the following example:

```
package main

import "fmt"

func printValue(v interface{}) {
    switch v := v.(type) {
    case string:
        fmt.Printf("%v is a string\n", v)
    case int:
        fmt.Printf("%v is an int\n", v)
    default:
        fmt.Printf("The type of v is unknown\n")
    }
}

func main() {
    v := 10
    printValue(v)
}
```

The running result is here:

```
10 is an int
```

Compared to `type assertion`, `type switch` uses keyword `type` instead of the specified variable type (such as `int`) in the parentheses.

References:
Effective Go;
Go – x.(T) Type Assertions;
How to find a type of a object in Golang?.

Types

Types in `Go` are divided into `2` categories: named and unnamed. Besides predeclared types (such as `int`, `rune`, etc), you can also define named type yourself. E.g.:

```
type mySlice []int
```

Unnamed types are defined by type literal. I.e., `[]int` is an unnamed type.

According to Go spec, there is an underlying type of every type:

> Each type T has an underlying type: If T is one of the predeclared boolean, numeric, or string types, or a type literal, the corresponding underlying type is T itself. Otherwise, T's underlying type is the underlying type of the type to which T refers in its type declaration.

So, in above example, both named type `mySlice` and unnamed type `[]int` have the same underlying type: `[]int`.

`Go` has strict rules of assigning values of variables. For example:

```
package main

import "fmt"

type mySlice1 []int
type mySlice2 []int

func main() {
    var s1 mySlice1
    var s2 mySlice2 = s1

    fmt.Println(s1, s2)
}
```

The compilation will complain the following error:

```
cannot use s1 (type mySlice1) as type mySlice2 in assignment
```

Although the underlying type of `s1` and `s2` are same: `[]int`, but they belong to `2` different named types (`mySlice1` and `mySlice2`), so they can't assign values each other. But if you modify `s2` 's type to `[]int`, the compilation will be OK:

```
package main

import "fmt"

type mySlice1 []int

func main() {
    var s1 mySlice1
    var s2 []int = s1

    fmt.Println(s1, s2)
}
```

The magic behind it is one rule of assignability:

> x's type V and T have identical underlying types and at least one of V or T is not a named type.

References:
Go spec;
Learning Go - Types;
Golang pop quiz.

io.Reader interface

`io.Reader` interface is a basic and very frequently-used interface:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

For every type who satisfies the `io.Reader` interface, you can imagine it's a pipe. Someone writes contents into one end of the pipe, and you can use `Read()` method which the type has provided to read content from the other end of the pipe. No matter it is a common file, a network socket, and so on. Only if it is compatible with `io.Reader` interface, I can read content of it.

Let's see an example:

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

func main() {
    file, err := os.Open("test.txt")
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    p := make([]byte, 4)
    for {
        if n, err := file.Read(p); n > 0 {
            fmt.Printf("Read %s\n", p[:n])
        } else if err != nil {
            if err == io.EOF {
                fmt.Println("Read all of the content.")
                break
            } else {
                log.Fatal(err)
            }
        } else /* n == 0 && err == nil */ {
            /* do nothing */
        }
    }
}
```

You can see after issuing a `read()` call, there are `3` scenarios need to be considered:

(1) `n > 0` : read valid contents; process it;
(2) `n == 0 && err != nil` : if `err` is `io.EOF` , it means all the content have been read, and there is nothing left; else something unexpected happened, need to do special operations;
(3) `n == 0 && err == nil` : according to io package document, it means nothing happened, so no need to do anything.

Create a `test.txt` file which only contains `5` bytes:

```
# cat test.txt
abcde
```

Executing the program, and the result is like this:

```
Read abcd
Read e
Read all of the content.
```

Reference:

io package document.

Decorate types to implement io.Reader interface

The io package has provided a bunch of handy read functions and methods, but unfortunately, they all require the arguments satisfy io.Reader interface. See the following example:

```
package main

import (
    "fmt"
    "io"
)

func main() {
    s := "Hello, world!"
    p := make([]byte, len(s))
    if _, err := io.ReadFull(s, p); err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%s\n", p)
    }
}
```

Compile above program and an error is generated:

```
read.go:11: cannot use s (type string) as type io.Reader in argument to io.ReadFull:
    string does not implement io.Reader (missing Read method)
```

The io.ReadFull function requires the argument should be compliance with `io.Reader` , but `string` type doesn't provide `Read()` method, so we need to do some tricks on `s` variable. Modify `io.ReadFull(s, p)` into `io.ReadFull(strings.NewReader(s), p)` :

```
package main

import (
    "fmt"
    "io"
    "strings"
)

func main() {
    s := "Hello, world!"
    p := make([]byte, len(s))
    if _, err := io.ReadFull(strings.NewReader(s), p); err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%s\n", p)
    }
}
```

This time, the compilation is OK, and the running result is:

```
Hello, world!
```

strings.NewReader function converts a `string` into a strings.Reader struct which supplies a read method:

```
func (r *Reader) Read(b []byte) (n int, err error)
```

Besides `string`, another common operation is to use bytes.NewReader to convert a byte slice into a bytes.Reader struct which satisfies `io.Reader` interface. Do some modifications on the above example:

```
 package main

import (
    "bytes"
    "fmt"
    "io"
    "strings"
)

func main() {
    s := "Hello, world!"
    p := make([]byte, len(s))
    if _, err := io.ReadFull(strings.NewReader(s), p); err != nil {
        fmt.Println(err)
    }

    r := bytes.NewReader(p)
    if b, err := r.ReadByte(); err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%c\n", b)
    }
}
```

`bytes.NewReader` converts the `p` slice into a `bytes.Reader` struct. The output is like this:

```
H
```

Buffered read

bufio package provides buffered read functions. Let's see an example:

(1) Create a `test.txt` file first:

```
# cat test.txt
abcd
efg
hijk
lmn
```

You can see `test.txt` contains `4` lines.

(2) See the following program:

```
package main

import (
	"bufio"
	"fmt"
	"io"
	"log"
	"os"
)

func main() {
	f, err := os.Open("test.txt")
	if err != nil {
		log.Fatal(err)
	}

	r := bufio.NewReader(f)
	for {
		if s, err := r.ReadSlice('\n'); err == nil || err == io.EOF {
			fmt.Printf("%s", s)
			if err == io.EOF {
				break
			}
		} else {
			log.Fatal(err)
		}

	}
}
```

(a)

```
f, err := os.Open("test.txt")
```

Open `test.txt` file.

(b)

```
r := bufio.NewReader(f)
```

`bufio.NewReader(f)` creates a bufio.Reader struct which implements buffered read function.

(c)

```
for {
    if s, err := r.ReadSlice('\n'); err == nil || err == io.EOF {
        fmt.Printf("%s", s)
        if err == io.EOF {
            break
        }
    } else {
        log.Fatal(err)
    }

}
```

Read and print each line.

The running result is here:

```
abcd
efg
hijk
lmn
```

We can also use bufio.Scanner to implement above "print each line" function:

```
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
)

func main() {
        f, err := os.Open("test.txt")
        if err != nil {
                log.Fatal(err)
        }

        s := bufio.NewScanner(f)

        for s.Scan() {
                fmt.Println(s.Text())
        }
}
```

(a)

```
s := bufio.NewScanner(f)
```

`bufio.NewScanner(f)` creates a new bufio.Scanner struct which splits the content by line by default.

(b)

```
for s.Scan() {
    fmt.Println(s.Text())
}
```

`s.Scan()` advances the `bufio.Scanner` to the next token (in this case, it is one optional carriage return followed by one mandatory newline), and we can use `s.Text()` function to get the content.

We can also customize SplitFunc function which doesn't separate content by line. Check the following code:

```
package main

import (
	"bufio"
	"fmt"
	"log"
	"os"
)

func main() {
	f, err := os.Open("test.txt")
	if err != nil {
		log.Fatal(err)
	}

	s := bufio.NewScanner(f)
	split := func(data []byte, atEOF bool) (advance int, token []byte, err error)
{
		for i := 0; i < len(data); i++ {
			if data[i] == 'h' {
				return i + 1, data[:i], nil
			}
		}

		return 0, data, bufio.ErrFinalToken
	}
	s.Split(split)
	for s.Scan() {
		fmt.Println(s.Text())
	}
}
```

The `split` function separates the content by " `h` ", and the running result is:

```
abcd
efg

ijk
lmn
```

io.Writer interface

The inverse of io.Reader is io.Writer interface:

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

Compared to `io.Reader`, since you no need to consider `io.EOF` error, the process of `Write` method is simple:

(1) `err == nil`: All the data in `p` is written successfully;
(2) `err != nil`: The data in `p` is partially or not written at all.

Let's see an example:

```
package main

import (
        "log"
        "os"
)

func main() {
        f, err := os.Create("test.txt")
        if err != nil {
                log.Fatal(err)
        }
        defer f.Close()

        if _, err = f.Write([]byte{'H', 'E', 'L', 'L', 'O'}); err != nil {
                log.Fatal(err)
        }
}
```

After executing the program, the `test.txt` is created:

```
# cat test.txt
HELLO
```

Check data race

"Data race" is a common but notorious issue in concurrency programs. sometimes it is difficult to debug and reproduce, especially in some big system, so this will make people very frustrated. Thankfully, the `Go` toolchain provides a `race detector` (now only works on `amd64` platform.) which can help us quickly spot and fix this kind of issue, and this can save our time even lives!

Take the following classic "data race" program as an example:

```
package main

import (
	"fmt"
	"sync"
)

var global int
var wg sync.WaitGroup

func count() {
	defer wg.Done()
	for i := 0; i < 10000; i++{
		global++
	}
}

func main() {
	wg.Add(2)
	go count()
	go count()
	wg.Wait()
	fmt.Println(global)
}
```

Two tasks increase `global` variable simultaneously, so the final value of `global` is non-deterministic. Using `race detector` to check it:

```
# go run -race race.go
==================
WARNING: DATA RACE
Read by goroutine 7:
  main.count()
      /root/gocode/src/race.go:14 +0x6d

Previous write by goroutine 6:
  main.count()
      /root/gocode/src/race.go:14 +0x89

Goroutine 7 (running) created at:
  main.main()
      /root/gocode/src/race.go:21 +0x6d

Goroutine 6 (running) created at:
  main.main()
      /root/gocode/src/race.go:20 +0x55
==================
19444
Found 1 data race(s)
exit status 66
```

Cool! the `race detector` finds the issue precisely, and it also provides the detailed tips of how to modifying it. Adding the lock of writing the `global` variable:

```
package main

import (
    "fmt"
    "sync"
)

var global int
var wg sync.WaitGroup
var w sync.Mutex

func count() {
    defer wg.Done()
    for i := 0; i < 10000; i++{
        w.Lock()
        global++
        w.Unlock()
    }
}

func main() {
    wg.Add(2)
    go count()
    go count()
    wg.Wait()
    fmt.Println(global)
}
```

This time, race detector is calm:

```
# go run -race non_race.go
20000
```

Please be accustomed to use this powerful tool frequently, you will appreciate it, I promise!

Reference:
Introducing the Go Race Detector.

Sort

`sort` package defines an interface whose name is `Interface`:

```
type Interface interface {
        // Len is the number of elements in the collection.
        Len() int
        // Less reports whether the element with
        // index i should sort before the element with index j.
        Less(i, j int) bool
        // Swap swaps the elements with indexes i and j.
        Swap(i, j int)
}
```

For slice, or any other collection types, provided that it implements the `Len()`, `Less` and `Swap` functions, you can use `sort.Sort()` function to arrange the elements in the order.

Let's see the following example:

```
package main

import (
    "fmt"
    "sort"
)

type command struct  {
    name string
}

type byName []command

func (a byName) Len() int           { return len(a) }
func (a byName) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a byName) Less(i, j int) bool { return a[i].name < a[j].name }


func main() {
    c := []command{
        {"breakpoint"},
        {"help"},
        {"args"},
        {"continue"},
    }
    fmt.Println("Before sorting: ", c)
    sort.Sort(byName(c))
    fmt.Println("After sorting: ", c)
}
```

To avoid losing focus of demonstrating how to use `sort.Interface`, the `command` struct is simplified to only contain one `string` member: `name`. The comparison method (`Less`) is just contrasting the `name` in alphabetic order.

Check the running result of the program:

```
Before sorting:  [{breakpoint} {help} {args} {continue}]
After sorting:  [{args} {breakpoint} {continue} {help}]
```

We can see after sorting, the items in `c` are rearranged.

Additionally, if you pick at the performance, you may define a slice whose type is the pointer, because switching pointer is much quicker if the size of element is very big. Modify the above example:

```
package main

import (
    "fmt"
    "sort"
)

type command struct  {
    name string
    help string
}

type byName []*command

func (a byName) Len() int           { return len(a) }
func (a byName) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a byName) Less(i, j int) bool { return a[i].name < a[j].name }

func main() {
    c := []*command{
        {"breakpoint", "Set breakpoints"},
        {"help", "Show help"},
        {"args", "Print arguments"},
        {"continue", "Continue"},
    }
    fmt.Println("Before sorting: ", c)
    sort.Sort(byName(c))
    fmt.Println("After sorting: ", c)
}
```

Check the executing result:

```
Before sorting:  [0xc0820066a0 0xc0820066c0 0xc0820066e0 0xc082006700]
After sorting:  [0xc0820066e0 0xc0820066a0 0xc082006700 0xc0820066c0]
```

You can see the pointers are reordered.

Reference:
The Go Programming Language.

range

`for ... range` clause can be used to iterate `5` types of variables: array, slice, string, map and channel, and the following sheet gives a summary of the items of `for ... range` loops:

Type	1st item	2nd item
Array	index	value
Slice	index	value
String	index (rune)	value (rune)
Map	key	value
Channel	value	

For array, slice, string and map, if you don't care about the second item, you can omit it. E.g.:

```
package main

import "fmt"

func main() {
    m := map[string]struct{} {
        "alpha": struct{}{},
        "beta": struct{}{},
    }
    for k := range m {
        fmt.Println(k)
    }
}
```

The running result is like this:

```
alpha
beta
```

Likewise, if the program doesn't need the first item, a blank identifier should occupy the position:

```
package main

import "fmt"

func main() {
    for _, v := range []int{1, 2, 3} {
        fmt.Println(v)
    }
}
```

The output is:

```
1
2
3
```

For channel type, the `close` operation can cause `for ... range` loop exit. See the following code:

```
package main

import "fmt"

func main() {
    ch := make(chan int)
    go func(chan int) {
        for _, v := range []int{1, 2} {
            ch <- v
        }
        close(ch)
    }(ch)

    for v := range ch {
        fmt.Println(v)
    }
    fmt.Println("The channel is closed.")
}
```

Check the outcome:

```
1
2
The channel is closed.
```

We can see `close(ch)` statement in another goroutine make the loop in main goroutine end.

Debugging

No one can write bug-free code, so debugging is a requisite skill for every software engineer. Here are some tips about debugging `Go` programs:

(1) Print
Yes! Printing logs seems the easiest method, but it is indeed the most effective approach in most cases. `Go` has provided a big family of printing functions in fmt package, and using them neatly is an expertise you should grasp.

(2) Debugger
In some scenarios, maybe you need the specialized debugger tools to help you spot the root cause. You can use `gdb`, but since "`GDB does not understand Go programs well.`" (from here), I suggest taking Delve, a dedicated debugger for `Go`, instead.

No matter using `gdb` or `Delve`, if you want to debug the executable file, you must pass `-gcflags "-N -l"` during compiling binary to disable code optimization, else some weird things can happen during debugging, such as you can't print the value of an already declared variable.

Except debugging the precompiled file, `Delve` can compile and debug the code on the fly, see the following example:

```
package main

import "fmt"

func main() {
    ch := make(chan int)
    go func(chan int) {
        for _, v := range []int{1, 2} {
            ch <- v
        }
        close(ch)
    }(ch)

    for v := range ch {
        fmt.Println(v)
    }
    fmt.Println("The channel is closed.")
}
```

Debugging flow is like this:

```
# dlv debug channel.go
Type 'help' for list of commands.
(dlv) b channel.go:14
Breakpoint 1 set at 0x401079 for main.main() ./channel.go:14
(dlv) c
> main.main() ./channel.go:14 (hits goroutine(1):1 total:1) (PC: 0x401079)
     9:                         ch <- v
    10:                 }
    11:                 close(ch)
    12:         }(ch)
    13:
=>  14:         for v := range ch {
    15:                 fmt.Println(v)
    16:         }
    17:         fmt.Println("The channel is closed.")
    18: }
(dlv) n
> main.main() ./channel.go:15 (PC: 0x4010c9)
    10:                 }
    11:                 close(ch)
    12:         }(ch)
    13:
    14:         for v := range ch {
=>  15:                 fmt.Println(v)
    16:         }
    17:         fmt.Println("The channel is closed.")
    18: }
(dlv) p v
1
(dlv) goroutine
Thread 12380 at ./channel.go:15
Goroutine 1:
        Runtime: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x42a283)
        User: ./channel.go:15 main.main (0x4010c9)
        Go: /usr/local/go/src/runtime/asm_amd64.s:145 runtime.rt0_go (0x453321)
(dlv) goroutines
[5 goroutines]
* Goroutine 1 - User: ./channel.go:15 main.main (0x4010c9)
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x42a283)
  Goroutine 3 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x42a283)
  Goroutine 4 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x42a283)
  Goroutine 5 - User: ./channel.go:9 main.main.func1 (0x4013a8)
(dlv)
```

Compared with `gdb`, `Delve` doesn't provide `start` command, so you need to set breakpoint first, then run `continue` command. You can see, `Delve` provides fruitful commands, e.g., you can check every goroutine status, so I think you should practice it frequently, and you will love it!

Happy debugging!

Goroutine

A running `Go` program is composed of one or more goroutines, and each goroutine can be considered as an independent task. Goroutine and thread have many commonalities, such as: every goroutine(thread) has its private stack and registers; if the main goroutine(thread) exits, the program will exit, and so on. But on modern Operating System (E.g., `Linux`), the actual execution and scheduled unit is thread, so if a goroutine wants to become running, it must "attach" to a thread. Let's see an example:

```
package main

import (
    "time"
)

func main() {
    time.Sleep(1000 * time.Second)
}
```

What the program does is just sleeping for a while, not does anything useful. After launching it on `Linux`, use `Delve` to attach the running process and observe the details of it:

```
(dlv) threads
* Thread 1040 at 0x451f73 /usr/local/go/src/runtime/sys_linux_amd64.s:307 runtime.fute
x
  Thread 1041 at 0x451f73 /usr/local/go/src/runtime/sys_linux_amd64.s:307 runtime.fute
x
  Thread 1042 at 0x451f73 /usr/local/go/src/runtime/sys_linux_amd64.s:307 runtime.fute
x
  Thread 1043 at 0x451f73 /usr/local/go/src/runtime/sys_linux_amd64.s:307 runtime.fute
x
  Thread 1044 at 0x451f73 /usr/local/go/src/runtime/sys_linux_amd64.s:307 runtime.fute
x
```

We can see there are `5` threads of this process, let's confirm it by checking `/proc/1040/task/` directory:

```
# cd /proc/1040/task/
# ls
1040  1041  1042  1043  1044
```

Yeah, the thread information of `Delve` is right! Check the particulars of goroutines:

```
(dlv) goroutines
[4 goroutines]
  Goroutine 1 - User: /usr/local/go/src/runtime/time.go:59 time.Sleep (0x43e236)
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x426f73)
  Goroutine 3 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x426f73)
* Goroutine 4 - User: /usr/local/go/src/runtime/lock_futex.go:206 runtime.notetsleepg
(0x40b1ce)
```

There is only one `main` goroutine, what the hell of the other `3` goroutines? Actually, the other `3` goroutines are system goroutines, and you can refer related info here. The number of `main` goroutine is `1`, and you can inspect it:

```
(dlv) goroutine 1
Switched from 4 to 1 (thread 1040)
(dlv) bt
0  0x0000000000426f73 in runtime.gopark
   at /usr/local/go/src/runtime/proc.go:263
1  0x0000000000426ff3 in runtime.goparkunlock
   at /usr/local/go/src/runtime/proc.go:268
2  0x000000000043e236 in time.Sleep
   at /usr/local/go/src/runtime/time.go:59
3  0x0000000000401013 in main.main
   at ./gocode/src/goroutine.go:8
4  0x0000000000426b9b in runtime.main
   at /usr/local/go/src/runtime/proc.go:188
5  0x0000000000451000 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1998
```

Using `go` keyword can create and start a goroutine, see another case:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)

    go func(chan int) {
        var count int
        for {
            count++
            ch <- count
            time.Sleep(10 * time.Second)
        }
    }(ch)

    for v := range ch {
        fmt.Println(v)
    }
}
```

The `go func` statement spawns another goroutine which works as a producer; while the `main` goroutine behaves as a consumer. And the output should be:

```
1
2
......
```

Use `Delve` to check the goroutine aspects:

```
(dlv) goroutines
[6 goroutines]
  Goroutine 1 - User: ./gocode/src/goroutine.go:20 main.main (0x40106c)
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x429fc3)
  Goroutine 3 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x429fc3)
  Goroutine 4 - User: /usr/local/go/src/runtime/proc.go:263 runtime.gopark (0x429fc3)
  Goroutine 5 - User: /usr/local/go/src/runtime/time.go:59 time.Sleep (0x442ab6)
* Goroutine 6 - User: /usr/local/go/src/runtime/lock_futex.go:206 runtime.notetsleepg
(0x40cf4e)
(dlv) goroutine 1
Switched from 6 to 1 (thread 1997)
(dlv) bt
0  0x0000000000429fc3 in runtime.gopark
   at /usr/local/go/src/runtime/proc.go:263
1  0x000000000042a043 in runtime.goparkunlock
   at /usr/local/go/src/runtime/proc.go:268
2  0x00000000004047eb in runtime.chanrecv
   at /usr/local/go/src/runtime/chan.go:470
3  0x0000000000404354 in runtime.chanrecv2
   at /usr/local/go/src/runtime/chan.go:360
4  0x000000000040106c in main.main
   at ./gocode/src/goroutine.go:20
5  0x0000000000429beb in runtime.main
   at /usr/local/go/src/runtime/proc.go:188
6  0x0000000000455de0 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1998
(dlv) goroutine 5
Switched from 1 to 5 (thread 1997)
(dlv) bt
0  0x0000000000429fc3 in runtime.gopark
   at /usr/local/go/src/runtime/proc.go:263
1  0x000000000042a043 in runtime.goparkunlock
   at /usr/local/go/src/runtime/proc.go:268
2  0x0000000000442ab6 in time.Sleep
   at /usr/local/go/src/runtime/time.go:59
3  0x00000000004011d6 in main.main.func1
   at ./gocode/src/goroutine.go:16
4  0x0000000000455de0 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1998
```

The number of `main` goroutine is `1`, whilst `func` is `5`.

Another caveat you should pay attention to is the switch point among goroutines. It can be blocking system call, channel operations, etc.

Reference:
Effective Go;
Performance without the event loop;
How Goroutines Work.

Functional literals

A functional literal just represents an anonymous function. You can assign functional literal to a variable:

```
package main

import (
    "fmt"
)

func main() {
    f := func() { fmt.Println("Hello, 中国！") }
    f()
}
```

Or invoke functional literal directly (Please notice the `()` at the end of functional literal):

```
package main

import (
    "fmt"
)

func main() {
    func() { fmt.Println("Hello, 中国！") }()
}
```

The above `2` programs both output "`Hello, 中国！`".

Functional literal is also a closure, so it can access the variables of its surrounding function. Check the following example which your real intention is `1` and `2` are printed:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 1; i <= 2; i++ {
        go func() {fmt.Println(i)}()
    }
    time.Sleep(time.Second)
}
```

But the output is:

```
3
3
```

The cause is the `func` goroutines don't get the opportunity to run until the `main` goroutine sleeps, and at that time, the variable `i` has been changed to `3`. Modify the above program as follows:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 1; i <= 2; i++ {
        go func() {fmt.Println(i)}()
        time.Sleep(time.Second)
    }
}
```

The `func` goroutine can run before `i` is changed, so the running result is what you expect:

```
1
2
```

But the idiom method should be passing `i` as an argument of the functional literal:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 1; i <= 2; i++ {
        go func(i int) {fmt.Println(i)}(i)
    }
    time.Sleep(time.Second)
}
```

In above program, When " `go func(i int) {fmt.Println(i)}(i)` " is executed (Note: not goroutine is executed.), `i` defined in `main()` is assigned to `func` 's local parameter `i` . And the result is:

```
1
2
```

P.S. You should notice, If you pass an argument while not use it, the `Go` compiler doesn't complain, but the closure will use the variable inherited from the parent function. That means the following statement:

```
go func(int) {fmt.Println(i)}(i)
```

equals to:

```
go func() {fmt.Println(i)}()
```

References:
The Go Programming Language Specification;
A question about passing arguments to closure;
Why add “()” after closure body in Golang?.

defer

The `defer` statement is used to postpone a function call executed immediately before the surrounding function returns. The common uses of `defer` include releasing resources (i.e., unlock the mutex, close file handle.), do some tracing(i.e., record the running time of function), etc. E.g., an ordinary accessing global variable exclusively is like this:

```
var mu sync.Mutex
var m = make(map[string]int)

func lookup(key string) int {
    mu.Lock()
    v := m[key]
    mu.Unlock()
    return v
}
```

An equivalent but concise format using `defer` is as follow:

```
var mu sync.Mutex
var m = make(map[string]int)

func lookup(key string) int {
    mu.Lock()
    defer mu.Unlock()
    return m[key]
}
```

You can see this style is more simpler and easier to comprehend.

The `defer` statements are executed in Last-In-First-Out sequence, which means the functions in latter `defer` statements run before their previous buddies. Check the following example:

```
package main

import "fmt"

func main()  {
    defer fmt.Println("First")
    defer fmt.Println("Last")
}
```

The running result is here:

```
Last
First
```

Although the function in `defer` statement runs very late, the parameters of the function are evaluated when the `defer` statement is executed.

```
package main

import "fmt"

func main()  {
    i := 10
    defer fmt.Println(i)
    i = 100
}
```

The running result is here:

```
10
```

Besides, if the deferred function is a function literal, it can also modify the return value:

```
package main

import "fmt"

func modify() (result int) {
    defer func(){result = 1000}()
    return 100
}

func main()  {
    fmt.Println(modify())
}
```

The value printed is `1000`, not `100`.

References:
The Go Programming Language;
Defer, Panic, and Recover.

error vs errors

Handling errors is a crucial part of writing robust programs. When scanning the `Go` packages, it is not rare to see APIs which have multiple return values with an error among them. For example:

> func Open(name string) (*File, error)
>
> Open opens the named file for reading. If successful, methods on the returned file can be used for reading; the associated file descriptor has mode O_RDONLY. If there is an error, it will be of type *PathError.

And the idiomatic method of using `os.Open` function is like this:

```
file, err := os.Open("file.go") // For read access.
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

So to implement resilient `Go` programs, how to generate and deal with errors is a required course.

`Go` provides both `error` and `errors`, and you shouldn't mix up them. `error` is a built-in interface type:

```
type error interface {
    Error() string
}
```

So for any type, as long as it implements `Error() string` method, it will satisfy `error` interface automatically. `errors` is one of my favorite packages since it is very simple (The life will definitely be easier if every package is similar to `errors`!). Removing the comments, the amount of core code lines is very small:

```
package errors

func New(text string) error {
    return &errorString{text}
}

type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

The `New` function in `errors` package returns an `errorString` struct which complies with `error` interface. Check the following example:

```
package main

import (
    "errors"
    "fmt"
)

func maxElem(s []int) (int, error) {
    if len(s) == 0 {
        return 0, errors.New("The slice must be non-empty!")
    }

    max := s[0]
    for _, v := range s[1:] {
        if v > max {
            max = v
        }
    }
    return max, nil
}

func main() {
    s := []int{}
    _, err := maxElem(s)
    if err != nil {
        fmt.Println(err)
    }
}
```

The execution result is here:

```
The slice must be non-empty!
```

In real life, you may prefer to use `Errorf` function defined in `fmt` package to create `error` interface, rather than use `errors.New()` directly:

> func Errorf(format string, a ...interface{}) error
>
> Errorf formats according to a format specifier and returns the string as a value that satisfies error.

So the above code can be refactored as follows:

```
func maxElem(s []int) (int, error) {
    ......
    if len(s) == 0 {
        return 0, fmt.Errorf("The slice must be non-empty!")
    }
    ......
}
```

References:
The Go Programming Language.

Send and receive operations on channel

`Go`'s built-in `channel` type provides a handy method for communicating and synchronizing: The producer pushes data into channel and the consumer pulls data from it.

The send operation on channel is simple, as long as the filled-in stuff is a valid expression and matches the channel's type:

```
Channel <- Expression
```

Take the following code as an example:

```
package main

func send() int {
    return 2
}
func main()  {
    ch := make(chan int, 2)
    ch <- 1
    ch <- send()
}
```

Receive operation on channel pulls the value from the channel, and you can save it or discard it if you don't care what you have got. Check the following example:

```
package main

import "fmt"

func main()  {
    ch := make(chan int)
    go func(ch chan int) {
        ch <- 1
        ch <- 2
    }(ch)
    <-ch
    fmt.Println(<-ch)
}
```

The running result is `2`, and that's because the first value (`1`) is left out in `<-ch` statement.

Compared to its send sibling, the receive operation is a little tricky: in assignment and initialization, there will be another return value which indicates whether this communication is successful or not. And the idiom of this variable's name is `ok`:

```
v, ok := <- ch
```

The value of `ok` is `true` if the value received was delivered by a successful send operation to the channel, or `false` if it is a zero value generated because the channel is closed and empty. That means although the channel is closed, as long as there is still data in the channel, the receive operation can of course get things from it. See the following code:

```
package main

import "fmt"

func main()  {
    ch := make(chan int)

    go func(ch chan int) {
        ch <- 1
        ch <- 2
        close(ch)
    }(ch)

    for i := 1; i <= 3; i++ {
        v, ok := <- ch
        fmt.Printf("value is %d, ok is %v\n", v, ok)
    }
}
```

The executing result is like this:

```
value is 1, ok is true
value is 2, ok is true
value is 0, ok is false
```

We can see after `func` goroutine executes closing channel operation, the value of `v` got from channel is the zero value of integer type: `0`, and `ok` is `false`.

Reference:
[The Go Programming Language Specification](#).

Channel types

When declaring variables of channel type, the most common instances are like this (`T` is any valid type):

```
var v chan T
```

But you may also see examples as follows :

```
var v <-chan T
```

Or:

```
var v chan<- T
```

What the hell are the differences among these `3` definitions? The distinctions are here:

(1) `chan T` : The channel can receive and send `T` type data;
(2) `<-chan T` : The channel is **read-only**, which means you can **only receive** `T` type data from this channel;
(2) `chan<- T` : The channel is **write-only**, which means you can **only send** `T` type data to this channel.

The mnemonics is correlating them with channel operations:

```
v := <-ch  // Receive from ch, and assign value to v.
ch <- v    // Send v to channel ch.
```

`<-chan T` is similar to `v := <-ch` , so it is a receive-only channel, and it is the same as `chan<- T` and `ch <- v` .

Restricting a channel type (read-only or write-only) can let compiler do strict checks for you. For example:

```
package main

func f() (<-chan int) {
    ch := make(chan int)
    return ch
}

func main()  {
    r := f()
    r <- 1
}
```

The compilation generates following errors:

```
invalid operation: r <- 1 (send to receive-only type <-chan int)
```

Furthermore, the `<-` operator associates with the **leftmost** `chan` possible, i.e., `chan<- chan int` and `chan (<-chan int)` aren't equal: the previous is same as `chan<- (chan int)`, which defines a **write-only** channel whose data type is a channel who can receive and send `int` data; while `chan (<-chan int)` defines a **write-and-read** channel whose data type is a channel who can only receive `int` data.

References:
Channel types;
How to understand "<-chan" in declaration?.

Unbuffered and buffered channels

The channel is divided into two categories: unbuffered and buffered.

(1) Unbuffered channel
For unbuffered channel, the sender will block on the channel until the receiver receives the data from the channel, whilst the receiver will also block on the channel until sender sends data into the channel. Check the following example:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)

    go func(ch chan int) {
        fmt.Println("Func goroutine begins sending data")
        ch <- 1
        fmt.Println("Func goroutine ends sending data")
     }(ch)

    fmt.Println("Main goroutine sleeps 2 seconds")
    time.Sleep(time.Second * 2)

    fmt.Println("Main goroutine begins receiving data")
    d := <- ch
    fmt.Println("Main goroutine received data:", d)

    time.Sleep(time.Second)
}
```

The running result likes this:

```
Main goroutine sleeps 2 seconds
Func goroutine begins sending data
Main goroutine begins receiving data
Main goroutine received data: 1
Func goroutine ends sending data
```

After the `main` goroutine is launched, it will sleep immediately(" `Main goroutine sleeps 2 seconds` " is printed), and this will cause `main` goroutine relinquishes the `CPU` to the `func` goroutine(" `Func goroutine begins sending data` " is printed). But since the `main` goroutine is sleeping and can't receive data from the channel, so `ch <- 1` operation in `func` goroutine can't complete until `d := <- ch` in `main` goroutine is executed(The final `3` logs are printed).

(2) Buffered channel
Compared with unbuffered counterpart, the sender of buffered channel will block when there is **no** empty slot of the `channel`, while the receiver will block on the channel when it is empty. Modify the above example:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int, 2)

    go func(ch chan int) {
        for i := 1; i <= 5; i++ {
            ch <- i
            fmt.Println("Func goroutine sends data: ", i)
        }
        close(ch)
    }(ch)

    fmt.Println("Main goroutine sleeps 2 seconds")
    time.Sleep(time.Second * 2)

    fmt.Println("Main goroutine begins receiving data")
    for d := range ch {
        fmt.Println("Main goroutine received data:", d)
    }
}
```

The executing result is as follows:

```
Main goroutine sleeps 2 seconds
Func goroutine sends data:  1
Func goroutine sends data:  2
Main goroutine begins receiving data
Main goroutine received data: 1
Main goroutine received data: 2
Main goroutine received data: 3
Func goroutine sends data:  3
Func goroutine sends data:  4
Func goroutine sends data:  5
Main goroutine received data: 4
Main goroutine received data: 5
```

In this sample, since the channel has `2` slots, so the `func` goroutine will not block until it sends the third element.

P.S., "`make(chan int, 0)`" is equal to "`make(chan int)`", and it will create an unbuffered `int` channel too.

nil channel VS closed channel

The zero value of channel type is `nil`, and the send and receive operations on a `nil` channel will always block. Check the following example:

```
package main

import "fmt"

func main() {
	var ch chan int

	go func(c chan int) {
		for v := range c {
			fmt.Println(v)
		}
	}(ch)

	ch <- 1
}
```

The running result is like this:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send (nil chan)]:
main.main()
	/root/nil_channel.go:14 +0x64

goroutine 5 [chan receive (nil chan)]:
main.main.func1(0x0)
	/root/nil_channel.go:9 +0x53
created by main.main
	/root/nil_channel.go:12 +0x37
```

We can see the `main` and `func` goroutines are both blocked.

The `Go` 's built-in `close` function can be used to close the channel which must not be receive-only, and it should always be executed by sender, not receiver. Closing a `nil` channel will cause panic. See the following example:

```
package main

func main() {
        var ch chan int
        close(ch)
}
```

The running result is like this:

```
panic: close of nil channel

goroutine 1 [running]:
panic(0x456500, 0xc82000a170)
 /usr/local/go/src/runtime/panic.go:481 +0x3e6
main.main()
 /root/nil_channel.go:5 +0x1e
```

Furthermore, there are also some subtleties of operating an already-closed channel:

(1) Close an already channel also cause panic:

```
package main

func main() {
        ch := make(chan int)
        close(ch)
        close(ch)
}
```

The running result is like this:

```
panic: close of closed channel

goroutine 1 [running]:
panic(0x456500, 0xc82000a170)
 /usr/local/go/src/runtime/panic.go:481 +0x3e6
main.main()
 /root/nil_channel.go:6 +0x4d
```

(2) Send on a closed channel will also introduce panic:

```
package main

func main() {
        ch := make(chan int)
        close(ch)
        ch <- 1
}
```

The running result is like this:

```
panic: send on closed channel

goroutine 1 [running]:
panic(0x456500, 0xc82000a170)
 /usr/local/go/src/runtime/panic.go:481 +0x3e6
main.main()
 /root/nil_channel.go:6 +0x6c
```

(3) Receive on a closed channel will return the zero value for the channel's type without blocking:

```
package main

import "fmt"

func main() {
        ch := make(chan int)
        close(ch)
        fmt.Println(<-ch)
}
```

The executing result is like this:

```
0
```

The following is a summary of "`nil` channel VS closed channel":

Operation type	Nil channel	Closed channel
Send	Block	Panic
Receive	Block	Not block, return zero value of channel's type
Close	Panic	Panic

References:

Package builtin;
Is it OK to leave a channel open?;
The Go Programming Language Specification.

Select operation

`Go` 's `select` operation looks similar to `switch` , but it's dedicated to poll send and receive operations channels. Check the following example:

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch1 := make(chan int)
	ch2 := make(chan int)

	go func(ch chan int) { <-ch }(ch1)
	go func(ch chan int) { ch <- 2 }(ch2)

	time.Sleep(time.Second)

	for {
		select {
		case ch1 <- 1:
			fmt.Println("Send operation on ch1 works!")
		case <-ch2:
			fmt.Println("Receive operation on ch2 works!")
		default:
			fmt.Println("Exit now!")
			return
		}
	}
}
```

The running result is like this:

```
Send operation on ch1 works!
Receive operation on ch2 works!
Exit now!
```

The `select` operation will check which `case` branch can be run, that means the send or receive action can be executed successfully. If more than one `case` are ready now, the `select` will randomly choose one to execute. If no `case` is ready, but there is a `default` branch, then the `default` block will be executed, else the `select` operation will block. In

the above example, if the `main` goroutine doesn't sleep (`time.Sleep(time.Second)`), the other `2 func` goroutines won't obtain the opportunity to run, so only `default` block in `select` statement will be executed.

The `select` statement won't process `nil` channel, so if a channel used for receive operation is closed, you should mark its value as `nil` , then it will be kicked out of the selection list. So a common pattern of selection on multiple receive channels looks like this:

```
for ch1 != nil && ch2 != nil {
    select {
    case x, ok := <-ch1:
        if !ok {
            ch1 = nil
            break
        }
        ......
    case x, ok := <-ch2:
        if !ok {
            ch2 = nil
            break
        }
        ......
    }
}
```

References:
The Go Programming Language Specification;
breaking out of a select statement when all channels are closed;
Curious Channels.

Need not close every channel

You don't need to close channel after using it, and it can be recycled automatically by the garbage collector. The following quote is from The Go Programming Language:

> You needn't close every channel when you've finished with it. It's only necessary to close a channel when it is important to tell the receiving goroutines that all data have been sent. A channel that the garbage collector determines to be unreachable will have its resources reclaimed whether or not it is closed. (Don't confuse this with the close operation for open files. It is important to call the Close method on every file when you've finished with it.)

References:
Is it OK to leave a channel open?;
The Go Programming Language.

Processing JSON object

JSON is a commonly used and powerful data-interchange format, and `Go` provides a built-in json package to handle it. Let' see the following example:

```
package main

import (
    "encoding/json"
    "log"
    "fmt"
)

type People struct {
    Name string
    age int
    Career string `json:"career"`
    Married bool `json:",omitempty"`
}

func main()  {
    p := &People{
        Name: "Nan",
        age: 34,
        Career: "Engineer",
    }

    data, err := json.Marshal(p)
    if err != nil {
        log.Fatalf("JSON marshaling failed: %s", err)
    }
    fmt.Printf("%s\n", data)
}
```

And the execution result is shown as follows:

```
{"Name":"Nan","career":"Engineer"}
```

The Marshal function is used to serialize an interface into a `JSON` object. In our example, it encodes a `People` struct:

(1) The `Name` member is encoded as our expectation:

```
"Name":"Nan"
```

(2) Where is the `age` field? We can't find it in our result. The cause is only exported members of struct can be marshaled, so that means only the name whose first letter capitalized can be encoded into `JSON` object (In our example, you should use `Age` instead of `age`).

(3) The name of `Career` field is `career` , not `Career` :

```
"career":"Engineer"
```

That's because the following tag: `json:"career"` , which tells the `Marshal` function to use `career` in the `JSON` object.

(4) We also can't see `Married` in the result although it has been exported, the magic behind is the `json:",omitempty"` tag which tells `Marshal` function no need to encode this member if it uses the default value.

There is another Unmarshal function which is used to parse a `JSON` object. See the following example which extends from the above one:

```
package main

import (
    "encoding/json"
    "log"
    "fmt"
)

type People struct {
    Name string
    age int
    Career string `json:"career"`
    Married bool `json:",omitempty"`
}

func main()  {
    var p People
    data, err := json.Marshal(&People{Name: "Nan", age: 34, Career: "Engineer", Marrie
d: true})

    if err != nil {
        log.Fatalf("JSON marshaling failed: %s", err)
    }

    err = json.Unmarshal(data, &p)
    if err != nil {
        log.Fatalf("JSON unmarshaling failed: %s", err)
    }

    fmt.Println(p)
}
```

The running result is like this:

```
{Nan 0 Engineer true}
```

We can see the `JSON` object is decoded successfully.

Besides `Marshal` and `Unmarshal` functions, the `json` package also provides Encoder and Decoder structs which are used to process `JSON` object from stream. E.g., It is not uncommon to see code which handle `HTTP` likes this:

```
func postFunc(w http.ResponseWriter, r *http.Request) {
    ......

    if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    ......
}
```

Because the mechanism of both methods are similar, it is not necessary to overtalk `Encoder` and `Decoder` here.

References:
Package json;
The Go Programming Language.

Use sync.WaitGroup to synchronize goroutines

(This post is a modification edition of Use sync.WaitGroup in Golang).

sync.WaitGroup provides a goroutine synchronization mechanism, and used for waiting for a collection of goroutines to finish. In the internal of `sync.WaitGroup` struct, there is a `counter` which records how many goroutines need to be waited are living now.

`sync.WaitGroup` provides `3` methods: `Add` , `Done` and `Wait` . `Add` method is used to identify how many goroutines need to be waited, and it will add `counter` value. When a goroutine exits, it must call `Done` , and it will decrease `counter` value by `1` . The `main` goroutine blocks on `Wait` , once the `counter` becomes `0` , the `Wait` will return, and main goroutine can continue to run.

Let's see an example:

```
package main

import (
    "sync"
    "time"
    "fmt"
)

func sleepFun(sec time.Duration, wg *sync.WaitGroup) {
    defer wg.Done()
    time.Sleep(sec * time.Second)
    fmt.Println("goroutine exit")
}

func main() {
    var wg sync.WaitGroup

    wg.Add(2)
    go sleepFun(1, &wg)
    go sleepFun(3, &wg)
    wg.Wait()
    fmt.Println("Main goroutine exit")

}
```

Because the `main` goroutine need to wait `2` goroutines, so the argument for `wg.Add` is `2`. The execution result is like this:

```
goroutine exit
goroutine exit
Main goroutine exit
```